Optimization

Optimization refers to a class of computational problems of great practical and economic significance. A company might need to determine the shortest route by which a sales representative might travel between a specified set of geographic locations ("travelling salesman problem"). Or given that airline prices sometimes irrationally price travel between adjoining cities much higher than travel between rather distant locations, the problem might be modified to finding the order for visits that minimizes the travel cost. Manufacturers are greatly concerned to find the arrangement of printed circuit traces, for a given set of components, that minimizes the manufacturing cost of the circuit board[1]. And scientists and engineers of all persuasions are interested in determining the "best" values of theoretical parameters (fundamental constants, for example, or merely empirical fitting parameters) by minimizing the deviation between a mathematical model and measurements.

1. Nonlinear least squares fitting

In connection with the latter, we saw in the last chapter how we can represent a given function by a finite sum of standard functions (polynomials, sines and cosines, and wavelets are typical) with unknown coefficients. This was a *linear* fit, the unknown coefficients being determined by requiring that they minimize the error-weighted squared deviation from the data.

However, sometimes we must fit data by a function that depends *non*linearly on its parameters. Consider

$$f_k \approx F\left[1 + e^{\alpha(x_k - X)}\right]^{-1} . \tag{1}$$

Although the dependence on the parameter F is linear, that on the parameters α and X is decidedly nonlinear. Several strategies can be used in such cases. The simplest, when it is possible, is to transform the data, to make the dependence on the parameters linear. That is, we re-express Eq. 1 in the form

$$\log\left(\frac{F}{f_k} - 1\right) \approx \alpha\,(x_k - X)\ . \tag{2}$$

If, for example, the value of F were known in advance rather than having to be determined from the data themselves, this transformation would be feasible. We note that the transformed variable

$$y_k = \frac{F}{f_k} - 1$$

[1] These examples are concerned with minimizing cost—clearly, that is the same as maximizing profit. That is, to maximize we can minimize the negative of the function to be maximized.

is linear in αX and α, so a linear least squares fit will determine α, then X. But in Eq. 1 no transformation will render linear the dependence on all three parameters at once. (Nevertheless posing the problem in the form Eq. 2 might simplify the labor of determining the three parameters, so this avenue ought to be explored.)

Thus we are frequently confronted with having to minimize numerically a complicated function of several parameters. Let us denote these by θ_0 , θ_1 , … , θ_{N-1} , and denote their possible range of variation by **R**. Then we want to find those values of $\{ \theta\} \subset \mathsf{R}$ that minimize a positive function:

$$\chi^2 \left(\bar{\theta}_0 , \dots , \bar{\theta}_{N-1} \right) = \min_{\{\theta\} \subset \mathsf{R}} \chi^2 \left(\theta_0 , \dots \theta_{N-1} \right). \tag{3}$$

One way to accomplish the minimization uses calculus, *via* the method of *steepest descents*. The idea is to differentiate the function χ^2 with respect to each θ_k, and to set the resulting N equations equal to zero, solving for the $N\,\bar{\theta}$'s. This is generally a pretty tall order, hence various approximate, iterative techniques have been developed. The simplest just steps along in θ-space, along the direction of the local downhill gradient $-\nabla\chi^2$, until a minimum is found. Then a new gradient is computed, and a new minimum sought[2].

Aside from the labor of computing $-\nabla\chi^2$, steepest descents has three main drawbacks: first, it only works for continuously differentiable functions[3]. We sometimes want to optimize functions with end-point minima or discintinuous derivatives. Second, it only guarantees to find *a* minimum, not necessarily the absolute minimum—if a function has several local minima, steepest descents may not find the lowest. Worse, consider a function that has a minimum in the form of a steep-sided gulley that winds slowly downhill to a declivity—somewhat like the channel of a meandering river. A naive steepest descents routine will then spend all its time bouncing up and down the banks of the gulley, rather than proceeding along its bottom, since the steepest gradient is always nearly perpendicular to the line of the channel. And third, one must actually compute the derivatives either as closed-form functions or *via* interpolation.

2. Functions of one variable

Before discussing functions of several variables, let us consider minimizing a function of a single variable. Since we know how to find roots of functions this might be as simple as seeking roots of the first derivative, assuming we can calculate it. Or we could seek the minimum be something akin to binary search. Either way we need to *bracket* the minimum—that is, to find a set of points $a < b < c$ for which $f(b) < f(a)$ and $f(b) < f(c)$. Once we know where a minimum may be found, we need a search strategy. One of the most popular[4] is the *golden section* search. The idea is that we

[2] This is not by itself very useful. Useful modifications can be found in W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, *Numerical Recipes: the art of scientific computing* (Cambridge University Press, Cambridge, 1986), p. 301 ff.

[3] That is, functions whose first derivative is continuous.

[4] Press, *et al.*, *ibid.*, p. 277 ff.

guess a new point x lying in, say, (b, c), and determine whether it is smaller than $f(b)$. If so, our new bounding set is $b < x < c$ and we iterate. Otherwise the new bounding set is (a, b, x). (And we iterate in that case.)

We want to guess x so as to reduce the interval by the largest amount possible on each iteration. Hence we pick it to lie in the larger of the two intervals, (a, b) or (b, c). Now suppose the point b lies a fraction w from the left end of the interval, *i.e.*

$$w = \frac{b-a}{c-a}, \qquad 1-w = \frac{c-b}{c-a};$$

and that the new guess lies a fraction z beyond b:

$$z = \frac{x-b}{c-a}.$$

Then the new bounding interval will be $w+z$ wide (if it is $a < x < b$) or $1-w$ wide in the other case. So we choose z to make these equal:

$$z = 1-2w.$$

Now where did w come from? We may suppose it arose in a previous iteration, in which case it was the optimal choice by this rule. That is, if z is optimal now, it must be the same fraction of the distance from b to c (assuming that was the larger segment) as b was to the distance from a to c. That is, we must have

$$\frac{z}{1-w} \equiv \frac{1-2w}{1-w} = w$$

or

$$w^2 - 3w + 1 = 0. \tag{4}$$

The solution of Eq. 4 that is smaller than 1 is

$$w = \frac{3-\sqrt{5}}{2} \approx 0.38196 \ldots .$$

This is the Golden Mean or Golden Section. The program to carry this out may be written in pseudocode as

```
\ Golden mean function minimizer, after
\ Press, et al., Numerical Recipes, 1st ed., p. 277.

\ say use( fn.name a b c )minimize

v: dummy                                  \ dummy function name

: )minimize   ( f: a b c epsilon -- x)  ( xt --)
    FLOCALS| epsilon cc bb aa |             \ local variable names
    defines dummy
    f" FMIN(aa,cc)"  f" cc = MAX(aa,cc)" aa F!     \ re-order limits
    BEGIN    new_x   rearrange_limits   converged?   UNTIL
   new_x  |FLOCALS  ;
```

Exercise
Convert the preceding pseudocode into a working program.

It is worth keeping in mind that (appropriately smooth) functions, in the vicinity of a minimum, look like

$$f(x) \approx f(x_0) + \frac{1}{2} f''(x_0)\,(x - x_0)^2 .$$

In that case, if ε is the relative precision of a floating point number (around 10^{-16} in IEEE double precision) we see that

$$2\,\varepsilon\,|f(x_0)| \approx \alpha^2\,|f''(x_0)|\,x_0^2$$

where α is the precision which we should specify for the location of the minimum. That is, we have

$$\alpha \approx \sqrt{\varepsilon} .$$

In words, we should not try to locate the position of the minimum withrelative precision better than $\sqrt{\varepsilon}$ lest we force the program into an endless loop.

3. Functions of several variables

Sometimes a function is so complex that its gradient is too expensive to compute. Can we find a minimum *without* evaluating partial derivatives? Several algorithms that do this have been devised. Here we explore two of them: the *simplex method* and *simulated annealing*.

The simplex method

The idea behind the simplex method is to construct a simplex—a set of $N+1$ distinct and non-degenerate[5] vertices in the N-dimensional θ-space. We evaluate the function f to be minimized at each of the vertices, and sort the table of vertices by the size of at each vertex, the best (smallest f) on top, the worst at the bottom. The simplex algorithm then chooses a new point in θ-space using a strategy that in action somewhat resembles the behavior of an amoeba seeking its food.

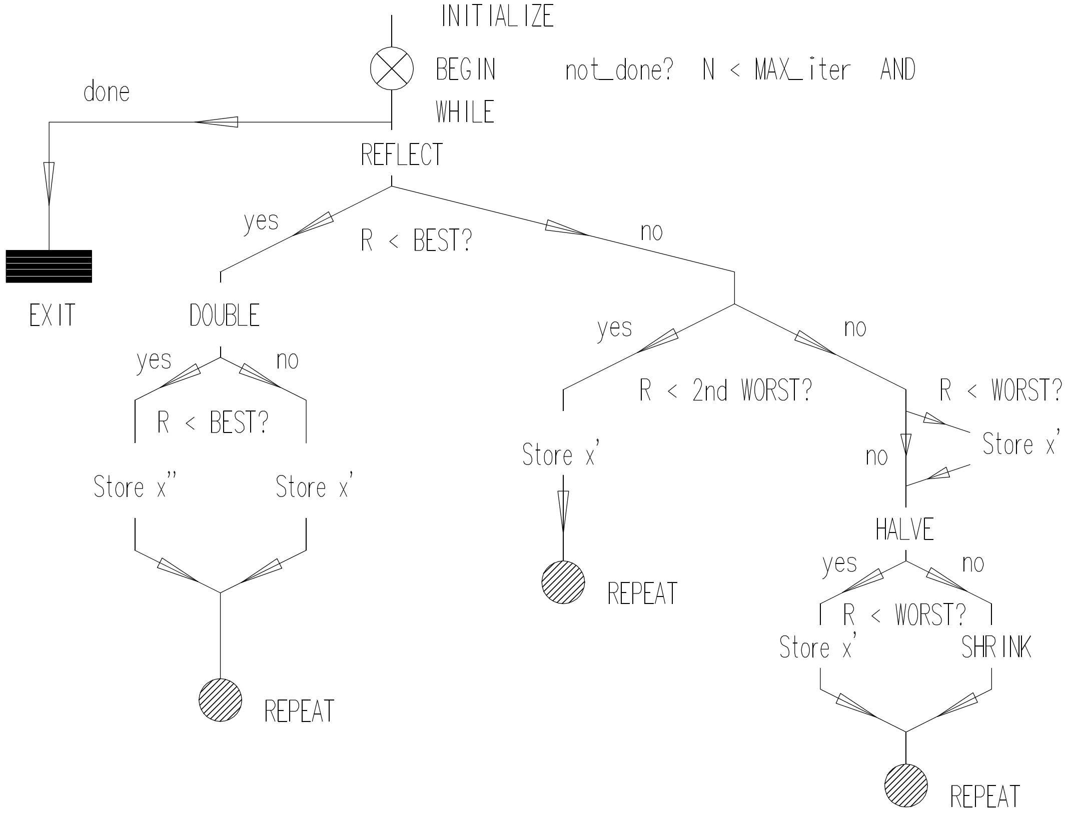
A standard Fortran subroutine for the simplex method has the disadvantages of being more than a page long and containing deeply nested control structures. It is thus hard to decipher—the corresponding flow chart (see p. 58 below) took some time to construct.

If the Fortran is translated directly to a more structured form in, say, Forth as shown below (with the various operations on the simplex factored out into subroutines), the indefinite outer loop (simulated

[5] "Non-degenerate" means the geometrical object, formed by connecting the $N+1$ vertices with straight lines, has non-zero N-dimensional volume; for example, if $N=2$, the simplex is a triangle.

```
     SUBROUTINE AMOEBA(P,Y,MP,NP,NDIM,FTOL,FUNK,ITER)
   C from Press, et al., "Numerical Recipes", p. 292.
     PARAMETER (NMAX=20,ALPHA=1.0,BETA=0.5,GAMMA=2.0,ITMAX=500)
     DIMENSION P(MP,NP),Y(MP),PR(NMAX),PRR(NMAX),PBAR(NMAX)
     MPTS=NDIM+1
     ITER=0
1    ILO=1
     IF(Y(1).GT.Y(2))THEN
       IHI=1
       INHI=2
     ELSE
       IHI=2
       INHI=1
     ENDIF
     DO 11 I=1,MPTS
       IF(Y(I).LT.Y(ILO)) ILO=I
       IF(Y(I).GT.Y(IHI))THEN
         INHI=IHI
         IHI=I
       ELSE IF(Y(I).GT.Y(INHI))THEN
         IF(I.NE.IHI) INHI=I
       ENDIF
11    CONTINUE
     RTOL=2.*ABS(Y(IHI)-Y(ILO))/(ABS(Y(IHI))+ABS(Y(ILO)))
     IF(RTOL.LT.FTOL)RETURN
     IF(ITER.EQ.ITMAX) PAUSE ’Amoeba exceeding maximum iterations.’
     ITER=ITER+1
     DO 12 J=1,NDIM
       PBAR(J)=0.
12    CONTINUE
     DO 14 I=1,MPTS
       IF(I.NE.IHI)THEN
         DO 13 J=1,NDIM
           PBAR(J)=PBAR(J)+P(I,J)
13        CONTINUE
       ENDIF
14    CONTINUE
     DO 15 J=1,NDIM
       PBAR(J)=PBAR(J)/NDIM
       PR(J)=(1.+ALPHA)*PBAR(J)-ALPHA*P(IHI,J)
15    CONTINUE
     YPR=FUNK(PR)
     IF(YPR.LE.Y(ILO))THEN
       DO 16 J=1,NDIM
         PRR(J)=GAMMA*PR(J)+(1.-GAMMA)*PBAR(J)
16      CONTINUE
       YPRR=FUNK(PRR)
       IF(YPRR.LT.Y(ILO))THEN
         DO 17 J=1,NDIM
           P(IHI,J)=PRR(J)
17        CONTINUE
         Y(IHI)=YPRR
       ELSE
         DO 18 J=1,NDIM
           P(IHI,J)=PR(J)
18        CONTINUE
```

```
         Y(IHI)=YPR
       ENDIF
     ELSE IF(YPR.GE.Y(INHI))THEN
       IF(YPR.LT.Y(IHI))THEN
         DO 19 J=1,NDIM
           P(IHI,J)=PR(J)
19        CONTINUE
         Y(IHI)=YPR
       ENDIF
       DO 21 J=1,NDIM
         PRR(J)=BETA*P(IHI,J)+(1.-BETA)*PBAR(J)
21      CONTINUE
       YPRR=FUNK(PRR)
       IF(YPRR.LT.Y(IHI))THEN
         DO 22 J=1,NDIM
           P(IHI,J)=PRR(J)
22        CONTINUE
         Y(IHI)=YPRR
       ELSE
         DO 24 I=1,MPTS
           IF(I.NE.ILO)THEN
             DO 23 J=1,NDIM
               PR(J)=0.5*(P(I,J)+P(ILO,J))
               P(I,J)=PR(J)
23            CONTINUE
             Y(I)=FUNK(PR)
           ENDIF
24        CONTINUE
       ENDIF
     ELSE
       DO 25 J=1,NDIM
         P(IHI,J)=PR(J)
25      CONTINUE
       Y(IHI)=YPR
     ENDIF
     GO TO 1
     END
```



in the Fortran routine by the penultimate **GOTO 1** statement) appears as a **BEGIN**… **WHILE**… **REPEAT** loop that is not nearly so long. Nevertheless, the triply nested **IF**… **ELSE**… **THEN**s make the control logic hard to follow.

```
: )MINIMIZE      INITIALIZE
   BEGIN   not_done?  N max_iter <  AND
   WHILE  REFLECT ( worst point thru geocenter of the rest to get x')
         x' Best_point   better?
         IF DOUBLE  ( find a point x'' twice as far from geocenter)
            x'' Best_point   better?
            IF store x'' ELSE   store x'  ENDIF
         ELSE
            x'  2nd_Worst_point   better?
            IF     store x'
            ELSE   x' Worst_point   better?
               IF  store x'  ENDIF
               HALVE  ( find x'' 0.5 as far from geocenter as REFLECTed pt)
               x'' Worst_point   better?
               IF     store x''
               ELSE   SHRINK  ( uniformly shrink all points toward Best_point)
               ENDIF
            ENDIF
         ENDIF
   REPEAT ;
```

The simplex algorithm attempts to find a point closer to the minimum than any of the current vertices of the simplex by moving away from the worst vertex (that with the highest value of the function).

First a new trial point is found by reflecting the worst vertex through the geometrical center of the other vertices (REFLECT). If that point is better than the best vertex so far, then the algorithm looks twice as far in the same direction (DOUBLE), the better of the two trial points replacing the former worst vertex. On the other hand, if the trial point found after REFLECTion is not better than the best, the algorithm inquires whether it is good enough to keep—that is, is it lower than the second-highest vertex. If so it replaces the worst vertex. What if it is not better than the second-worst vertex? Then the routine tests whether it is better than the worst. If so it replaces the worst, but before ending the outer loop, a new operation is performed: a trial point is found halfway between the old trial point and the geocenter of the others (HALVE). If this new point is an improvement over the worst point, it is stored; otherwise a last, desperate attempt to improve things is made: the lowest vertex is held fixed and the rest of the simplex is shrunk uniformly toward it by some scale factor.

A complex sequence of operations requiring multiple decisions often may be more clearly represented by a *finite state machine* than by a multiply branching binary logic tree. A state machine is the software analog of certain electromechanical devices[6] (such as the device that accepts coins in a vending machine, dispensing goods, making change, and rejecting slugs as necessary). State machines are often represented by graphs, but for programming purposes a tabular representation is clearer.

The simplex algorithm requires us to determine whether the trial vertex is better than the best; between the best and the second-worst; between the second-worst and the worst; or worse than the worst. That is, there are four possibilities that determine four courses of action. These can be represented as in the table below:

state \ input	R < best?	best<R<2worst?	2worst<R<worst?	worst<R?
reflected	store x′ DOUBLE → ×2	store x′ → exit	store x′ HALVE → ½	HALVE → ½
× 2	store x′′ → exit	noop → exit	noop → exit	noop → exit
½	noop → exit	noop → exit	store x′ → exit	SHRINK → exit
exit				

Each cell contains an action (or set of actions) and a state transition to be made after the action is taken. The column labels are inputs that must exhaust all possibilities, and the row labels represent the state the machine is in when presented with the input represented by the column label. The state

[6] Zvi Kohavi, *Switching and Finite Automata Theory*, 2nd ed. (McGraw-Hill Publishing Co, New York, 1978).

labelled "exit" is a terminal state, hence its cells contain neither actions nor transitions. The noop ("do nothing") action could, if one wished, be replaced by an error handler that would inform us if—say by hardware failure—the (software) finite state machine has landed in a cell that should be impossible to reach (these cells are indicated by being tinted).

The chief virtue of the state machine representation of complex logic is the impossibility of producing "dead" code that is impossible to reach. Moreover the state transition table shows explicitly which input leads to which action. With such a state machine (called slither below), the)MINIMIZE subroutine becomes

```
: )MINIMIZE      INITIALIZE
   BEGIN   not_done?  N max_iter <  AND
   WHILE  REFLECT   ( worst point thru geocenter of the rest to get x')
      reflected  >state slither  ( initialize FSM)
      BEGIN     test_x'          ( result is a column number, 0-3)
                slither          ( operate FSM)
                state: slither   ( get current state)
                exit =           ( test for exit state)
      UNTIL
   REPEAT  ;
```

which, when accompanied by the code for slither in tabular format, is much easier to follow than the previous version. Precisely how the state machine is implemented is up to the programmer. For example, a CASE… ENDCASE control structure would fit the bill. Forth is so versatile it has been possible to devise a method of compiling a tabular representation resembling that above directly into a subroutine[7]. This latter method is essentially self-documenting.

Simulated annealing

This method has become important in "solving" certain *NP-complete* programming tasks (this term means that the problem's running time scales like $e^{\lambda N}$ with the size of the problem). Although it might require a very long time to get an exact solution—for example the absolute minimum distance a travelling salesman must go, to visit N cities in a round trip, or the absolute minimum amount of wiring to connect up circuit elements on a printed circuit board—it is possible to come close to the true minimum in a much shorter time. Thus if one can be satisfied with an answer that is within—say—5% of optimum, the computing time can be quite brief.

Simulated annealing takes its name from thermodynamics. A chunk of glass might have many internal cracks and dislocations, and thus be in a state of greater energy than a similar chunk that has no cracks. Typically we heat the glass and slowly cool it to remove the cracks and internal strains.

[7] See, *e.g.*, J.V. Noble, "Avoid Decisions", *Computers in Physics* 5, #4 (1991) 386; J.V. Noble, *Finite State Machines in Forth* (http://www.jfar.org/article001.html).

From a thermodynamic point of view, the probability for the system to be in a state of energy E at temperature $\Theta = kT_{absolute}$ is

$$P \sim e^{-E/\Theta} .$$

The initial (cracked) state of the glass is a local minimum, although not the absolute minimum, of the energy. To get into another state—perhaps of lower energy—the system must pass through a "barrier" or intermediate stae of higher energy. This is very unlikely when the temperature is low. However, at higher temperatures the glass can with reasonable probability pass through many intermediate states of higher energy. Its most likely state is, of course, the lowest-energy one. Thus if the temperature is first raised (and the system allowed to equilibrate) then slowly lowered again, there is a good chance the sysem will be trapped in a state of lower energy, one in which the cracks and internal strains are virtually nonexistent.

In other words, to minimize a function $f\left(\alpha_1 , \dots , \alpha_n\right)$ we compute (randomly) a trial value of the vector $\vec{\alpha} = (\alpha_1 , \dots , \alpha_n)$ and compare $f(\vec{\alpha}_{\text{new}})$ with $f(\vec{\alpha}_{\text{old}})$. If the new value is smaller, then it replaces the old one. If it is larger, then a random number lying in the interval [0,1] is computed. If it is less than or equal to the transition probability

$$P = \exp\left[-(f_{\text{new}} - f_{\text{old}})/\Theta\right]$$

the new state is accepted, otherwise it is rejected.

This procedure is repeated many times, and at the same time the "temperature" Θ is gradually reduced. At some point very few transitions are taking place, at which point we decide the process has converged. To some extent it is useful to provide the feedback and interaction of a human programmer, since the best schedule for varying the "temperature" is by no means obvious and it is often a good idea to experiment. One should not feel a great deal of confidence in the result of a single run. It is usually valuable to run the program several times with different initial conditions to see whether it really finds minima close to the optimum.

Simulated annealing, because of its employment of random processes, belongs to the class of numerical techniques called Monte Carlo methods (after the famous gambling casinos of the Principality of Monaco). We discuss other Monte Carlo methods in Chapter 4.

4. Linear programming

Imagine you are a plant manager, and that your factory manufactures three products: floor tile, counter top, and wall tiles. The main raw materials are asphalt and vinyl, differing amounts in each unit of product. Labor and machine time are also required for each product. Suppose the inputs and outputs look like this[8]:

	Floor tile	**Counter top**	**Wall tile**	**Am't /day**
Vinyl	30	10	50	1500
Asphalt	5	0	3	200
Labor	0.2	0.1	0.5	12
Machine	0.1	0.2	0.3	9
Profit	10.00	5.00	5.50	?

If we call the amounts of floor tile, counter top and wall tile made per day x_0 , x_1 , x_2 then we can express the daily profit as

$$P(\mathbf{x}) = 10.00\,x_0 + 5.00x_1 + 5.50x_2 \,.$$

However, we cannot make as much of each as we would like, because the supplies of materials, labor and machine time are limited; that is,

$$30\,x_0 + 10\,x_1 + 50\,x_2 \le 1500$$

$$5\,x_0 + 0\,x_1 + 3\,x_2 \le 200$$

$$0.2\,x_0 + 0.1\,x_1 + 0.5\,x_2 \le 12$$

$$0.1\,x_0 + 0.2\,x_1 + 0.3\,x_2 \le 9$$

The first thing we realize is that the x's are all positive since we cannot make *less* than none of a product. Next we realize there are more inequalities than independent variables (x's). So it *may not be possible* to satisfy all of them simultaneously. Third, suppose we normalize the equations by dividing each through by its respective limit (right side of the inequality): then all the right sides become unity, and the left sides look like a matrix multiplying a vector:

$$\mathrm{A}\cdot x \le \widehat{e} \;,$$

where $\widehat{e}$ is a column vector, each of whose elements is 1.

Clearly, for each x the most restrictive inequality is the one with the *largest* (positive) coefficient for that x, after normalization. The largest coefficient of x_0 is $\frac{1}{40}$, occurring in the second inequality (limitation of asphalt). The largest for x_1 is $\frac{1}{45}$, from the fourth (limitation of machine time). And

[8] This example comes from E.D. Nering and A.W. Tucker, *Linear Programs and Related Problems* (Academic Press, Inc., San Diego, 1993).

the largest for x_2 is ½4, from the third inequality (limitation of labor). Hence we know the solution, if any, lies in the ranges

$$x_0 \in [0,40]$$
$$x_1 \in [0,45]$$
$$x_2 \in [0,24]$$

Mathematical description

Linear programming deals with the preceding kind of problem: a manufacturer's total profit is the sum of profits from producing x_0 of one item, x_1 of another, *etc*. That is, the total can be expressed as an *objective function* linear in **x**; the most general form for this function is obviously

$$P(\mathbf{x}) = a^{(0)} \cdot \mathbf{x} \,.$$

All the x_k's must be positive or zero—it is impossible to manufacture a negative quantity of something. Thus,

$$x_k \ge 0, \; k = 0, 1, \ldots, N-1 \,.$$

The costs of manufacturing the products, as well as constraints of availability and/or minimum requirements, for materials and labor, are expressed as inequalities (or equalities) of the form

$$\overrightarrow{a}^{(k)}\cdot\overrightarrow{x} \le b_k \,, \; k=1,\ldots,m_1 \,, \; (b_k \ge 0)$$

$$\overrightarrow{a}^{(k)}\cdot\overrightarrow{x} \ge b_k \ge 0, \quad k=m_1+1,\ldots,m_1+m_2$$

$$\overrightarrow{a}^{(k)}\cdot\overrightarrow{x} = b_k \ge 0$$
$$k = m_1+m_2+1,\ldots, m_1+m_2+m_3$$

The $M \equiv m_1 + m_2 + m_3$ constraints may be more or less numerous than the number N of unknowns.

As (mathematically inclined) businessmen we should like to find an N-dimensional vector $\overrightarrow{x}$ that maximizes our profit. Sometimes this means we maximize $P(\overrightarrow{x})$ subject to the above constraints, other times we minimize it. As an example of the latter, when we raise livestock using feeds containing different amounts of essential nutrients and varying in cost per unit of feed, we must feed each animal what it needs. The constraints for each essential nutrient are inequalities

$$\overrightarrow{a}^{(k)}\cdot\overrightarrow{x} \ge b_k \ge 0, \; x_j \ge 0,$$

which determine how we can feed our beasts as cheaply as possible.

Remarks and terminology

The literature of linear programming can be impenetrable. The admirable compendium *Numerical Recipes*[9] notes

"As you see, the subject of linear programming is surrounded by notational and terminological thickets. Both of these thorny defenses are lovingly cultivated by a coterie of stern acolytes who have devoted themselves to the field."

Consider the very name of the subject: it is obviously *linear*, but why call it *programming*? In fact the term belongs to an era antedating computers and computer programs. Its sense was "government program"—as applied in the Departments of Agriculture, Defense or Health and Human Resources.

A vector of non-negative components that satisfies the constraints is called *feasible*. If it also maximizes $P(\vec{x})$ it is called *optimal*. There might be no vector at all that solves the problem—for example, there might be no feasible vectors. Alternatively, the constraints might not constrain sufficiently, allowing one or more of the x_k's to be made infinite, thus making the objective function infinite. So a computer program to solve the problem must determine whether a solution actually exists while trying to compute it.

It was once thought that the general linear programming problem was NP-complete. And in fact the algorithm we expound here, the *simplex method*[9], has a worst-case running time that increases exponentially with the size of the problem[10]. In 1979, however, the Russian mathematician Khachiyan found an algorithm that runs in polynomial time[11]. Subsequently Karmarkar found a vastly more practical polynomial-time algorithm[12].

In that case, why bother to discuss the simplex method here? The answer is that for most problems the simplex method is much better than the worst case, *i.e.* it is reasonably efficient. Moreover it is much easier to program than Karmarkar's method, and for problems of moderate size is faster. The polynomial-time algorithm wins (and by factors of up to 50) for problems with upwards of 4000 variables, such as encountered in—say—economic models describing major industrial sectors or whole nations.

[9] The simplex algorithm was invented by G.B. Dantzig in 1947.

[10] This was shown by a family of examples constructed by V. Klee and G.J. Minty, "How good is the simplex algorithm?" *Inequalities-III*, O. Shisha, ed. (Academic Press, New York, 1972).

[11] L.G. Khachiyan, "A polynomial algorithm in linear programming", *Doklady Akademia Nauk SSSR* **224** (1979) 1093-1096.

[12] N.K. Karmarkar, "A new polynomial-time algorithm for linear programming", *Combinatoria* **4** (1984) 373-395. See also Byte Magazine, September 1987, for a popular exposition and BASIC programs implementing Karmarkar's algorithm.

The Simplex Algorithm

To see how the algorithm works, let us return to our example problem, whose inequalities have been (re)expressed as

$$\begin{pmatrix} 1/50 & 1/150 & 1/30 \\ 1/40 & 0 & 3/200 \\ 1/60 & 1/120 & 1/24 \\ 1/90 & 1/45 & 1/30 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \le \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

The first step is to introduce a new set of variables, y_k, so-called *slack variables*[13] that convert the inequalities to equalities:

$$y_0 = 1 - 1/50\, x_0 - 1/150\, x_1 - 1/30\, x_2$$

$$y_1 = 1 - 1/40\, x_0 - 0\, x_1 - 3/200\, x_2$$

$$y_2 = 1 - 1/60\, x_0 - 1/120\, x_1 - 1/24\, x_2$$

$$y_3 = 1 - 1/90\, x_0 - 1/45\, x_1 - 1/30\, x_2$$

The original linear programming problem has now been put in *restricted normal form*. (In fact, any such problem can be put into restricted normal form.) The next step is to write the system in tabular form:

		x_0	x_1	x_2
P	0	10	5	5.5
y_0	1	$-1/50$	$-1/150$	$-1/30$
y_1	1	$-1/40$	0	$-3/200$
y_2	1	$-1/60$	$-1/120$	$-1/24$
y_3	1	$-1/90$	$-1/45$	$-1/30$

The first line ("z-row") represents the objective function $P(\mathbf{x})$; the next four lines represent the constraints expressed as equalities. The y's in the starting tableau are called "left-hand variables" (*LHVs*), and the x's "right-hand variables"(*RHV*'s).

The simplex method proceeds very much like pivotal elimination: We choose a right-hand variable with a positive coefficient in P (because otherwise increasing it would make P smaller). Recall these coefficients are found in the z-row.

Next we look for the (constraint) row in the tableau with the least (most negative) coefficient under the chosen *RHV*. (This will produce the smallest value of that *RHV*.) Finally, we see which of such values of the selected *RHV*'s would produce the biggest increase in P.

[13] They are called *slack* variables because they "take up the slack". Note that all the y's are positive.

In the starting tableau all three *RHV*'s have positive coefficients in *P*—they all qualify.

		x_0	x_1	x_2
P	0	10	5	5.5
y_0	1	$-1/50$	$-1/150$	$-1/30$
y_1	1	$\boxed{-1/40}$	0	$-3/200$
y_2	1	$-1/60$	$-1/120$	$-1/24$
y_3	1	$-1/90$	$-1/45$	$-1/30$

The minimum values of x_0, x_1 and x_2 are, respectively, 40, 45 and 24. These produce corresponding changes of +400, +225 and +132 in *P*. Hence the pivot element is

$$a_{10} = -1/40 ,$$

shown boxed in the tableau. We then use row 1 to eliminate x_0 in favor of x_1, x_2 and y_1:

$$x_0 = 40\,(1 - y_1 - 3/200\, x_2).$$

This leads to the revised tableau:

		x_1	x_2	y_1
P	400	5	-0.5	-400
y_0	$1/5$	$-1/150$	?	$4/5$
y_2	$1/3$	$-1/120$	?	$2/3$
y_3	$5/9$	$-1/45$	?	$4/9$

Exercise

Fill in the coefficients of x_2 we have represented as question marks in the above tableau.

End of Exercise

Now we notice immediately that the coefficient in *P* of y_1 is negative, hence the best that can be done is to set y_1 to zero. We do this, giving the simplified tableau

		x_1	x_2
P	400	5	-0.5
y_0	$1/5$	$-1/150$	?
y_2	$1/3$	$-1/120$	?
y_3	$5/9$	$-1/45$	?

But the coefficient of x_2 in *P* is also negative. Thus we must set $x_2 = 0$. (That is why it was not so important to calculate its coefficients.)

Of the remaining qualifying (negative) coefficients, $-1/45$ is the least, hence we next eliminate x_1 using row 3:

$x_1 = 25(1 - y_3)$.

The resulting tableau is in its terminal form:

		y_3
P	525	-125
y_0	⅕	$-1/150$
y_2	⅓	$-1/120$

obviously $y_3 = 0$, $x_1 = 25$, and Bob's your uncle[14]. The maximum of P is 525, achieved with values

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 40 \\ 25 \\ 0 \end{pmatrix}.$$

Question

Is the above solution **feasible** (*i.e.* does it satisfy all the constraints)?

The simplex program

Let us recall the steps of the simplex algorithm:

- construct the starting tableau
- select a pivot element;
- express a selected right-hand variable in terms of its slack variable and remaining *RHV*'s;
- construct reduced tableau (with one fewer row);
- prune columns with negative coefficients in the *z*-row;
- iterate until done or until remaining constraints become infeasible.

[14] …as our British friends might say, meaning "See how easy *that* was?"

In pseudocode, we might express these steps as

```
: }}simplex ( adr --)
      tableau!     \ make tableau (in restricted normal form)
      BEGIN
         >pivot    \ find pivot element
         reduce    \ eliminate row with pivot element; re-express
                   \ all other rows in terms of the (eliminated) RHV
         prune     \ drop columns with negative coefficients in z-row
         done?
      UNTIL
;
```

What would we do if we were trying to *minimize* an objective function rather than maximize it? That is, suppose we were trying to solve the animal feed problem represented by the following tableau:

	protein	**carbo-hydrate**	**roughage**	**cost/wk**
hay	10	10	60	3.80
oats	15	10	25	5.00
pellets	10	5	55	3.50
sweet feed	25	20	35	8.00
total req.	450	400	1050	?

Our constraints now look like

$$10\,y_0 + 15\,y_1 + 10\,y_2 + 25\,y_3 \ge 450$$

$$10\,y_0 + 10\,y_1 + \ 5\,y_2 + 20\,y_3 \ge 400$$

$$60\,y_0 + 25\,y_1 + 55\,y_2 + 35\,y_3 \ge 1050$$

and we want to minimize

$$C(\mathbf{y}) = 3.8\,y_0 \ + \ 5.0\,y_1 \ + \ 3.5\,y_2 \ + \ 8.0\,y_3$$

To put the problem more formally, we want to minimize

$$C(\mathbf{y}) \overset{df}{=} \mathbf{c} \cdot \mathbf{y}$$

subject to the conditions

$$\mathbf{A} \cdot \mathbf{y} \ge \mathbf{b}, \quad \mathbf{y} \ge \mathbf{0}\,.$$

Conversely, the earlier problem was to maximize

$$P(\mathbf{x}) = \mathbf{b} \cdot \mathbf{x}$$

subject to conditions

$$\widetilde{\mathbf{A}} \cdot \boldsymbol{x} \le \mathbf{c}, \quad \mathbf{x} \ge \mathbf{0}\,,$$

where $\widetilde{\mathbf{A}}$ signifies the transpose of the matrix **A**.

We may note that

$$\mathbf{b}\cdot\mathbf{x} \le \mathbf{x}\cdot(\mathbf{A}\cdot\mathbf{y}) \equiv \mathbf{y}\cdot(\widetilde{\mathbf{A}}\cdot\mathbf{x}) \le \mathbf{c}\cdot\mathbf{y}$$

so that if we adjust the *y*'s to minimize C(**y**), or adjust the *x*'s to maximize *P*(**x**), the values must be equal if there is any solution at all. Hence a minimization problem with "greater-than" constraints can be converted into a maximization one with "less-than" constraints simply by interchanging the right- and left-hand variables of the problem.

The (maximization) tableau corresponding to the preceding livestock feeding problem becomes

		x_0	x_1	x_2
P	0	450	400	1050
y_0	3.8	-10	-10	-60
y_1	5.0	-15	-10	-25
y_2	3.5	-10	-5	-55
y_3	8.0	-25	-20	-35

Exercise

Translate the pseudocode subroutine `}}simplex` into a Forth routine that minimizes an objective function whose constraints are all expressed as upper bounds (as in the floor covering example above). Test it on the examples given above.

A Fortran subroutine for the simplex algorithm[15] is displayed on the next two pages, shorn of three subroutines that it calls. We quote it just to reiterate how hard it is to understand long subroutines, no matter how prettily displayed or well-commented.

[15] Press, *et al.*, *ibid.*, p. 322.

```
C      SIMPLEX ALGORITHM FOR LINEAR PROGRAMMING
C      -- FROM "NUMERICAL RECIPES" (SEE TEXT)
C      -- BASED ON THE IMPLEMENTATION OF
C         KUENZI, TSZCHACH AND ZENDER
C          "NUMERICAL METHODS OF MATHEMATICAL
C         OPTIMIZATION" (ACAD. PRESS, NY, 1971)
C
      SUBROUTINE SIMPLX(A,M,N,MP,NP,M1,M2,M3,ICASE,IZROV,IPOSV)
      PARAMETER(MMAX=100,EPS=1.E-6)
      DIMENSION A(MP,NP),IZROV(N),IPOSV(M),L1(MMAX),L2(MMAX),L3(MMAX)
      IF(M.NE.M1+M2+M3)PAUSE 'Bad input constraint counts.'
      NL1=N
      DO 11 K=1,N
        L1(K)=K
        IZROV(K)=K
 11   CONTINUE
      NL2=M
      DO 12 I=1,M
        IF(A(I+1,1).LT.0.)PAUSE 'Bad input tableau.'
        L2(I)= I
        IPOSV(I)= N+I
 12    CONTINUE
      DO 13 I= 1,M2
        L3(I)= 1
 13    CONTINUE
      IR = 0
      IF(M2+M3.EQ.0)GO TO 30
      IR=1
      DO 15 K=1,N+1
        Q1=0.
        DO 14 I= M1+1,M
          Q1= Q1+A(I+1,K)
 14     CONTINUE
        A(M+2,K)=-Q1
 15   CONTINUE
 10   CALL SIMP1(A,MP,NP,M+1,L1,NL1,0,KP,BMAX)
      IF(BMAX.LE.EPS.AND.A(M+2,1).LT.-EPS)THEN
        ICASE= -1
        RETURN
      ELSE IF(BMAX.LE.EPS.AND.A(M+2,1).LE.EPS)THEN
        M12 = M1+M2+1
        IF(M12.LE.M)THEN
          DO 16 IP = M12,M
            IF(IPOSV(IP).EQ.IP+N)THEN
              CALL SIMP1(A,MP,NP,IP,L1,NL1,1,KP,BMAX)
              IF(BMAX.GT.0.)GO TO 1
            ENDIF
 16       CONTINUE
        ENDIF
        IR = 0
        M12 = M12-1
        IF(M1+1.GT.M12)GO TO 30
        DO 18 I =M1+1,M12
          IF(L3(I-M1).EQ.1)THEN
            DO 17 K = 1,N+1
              A(I+1,K)=-A(I+1,K)
17          CONTINUE
          ENDIF
18      CONTINUE
        GO TO 30
      ENDIF
      CALL SIMP2(A,M,N,MP,NP,L2,NL2,IP,KP,Q1)
      IF(IP.EQ.0)THEN
        ICASE = -1
        RETURN
      ENDIF
1      CALL SIMP3(A,MP,NP,M+1,N,IP,KP)
      IF(IPOSV(IP).GE.N+M1+M2+1)THEN
        DO 19 K = 1,NL1
          IF(L1(K).EQ.KP)GO TO 2
19      CONTINUE
2        NL1=NL1-1
        DO 21 IS = K,NL1
          L1(IS)=L1(IS+1)
21      CONTINUE
      ELSE
        IF(IPOSV(IP).LT.N+M1+1)GO TO 20
        KH = IPOSV(IP)-M1-N
        IF(L3(KH).EQ.0)GO TO 20
        L3(KH) = 0
      ENDIF
      A(M+2,KP+1) = A(M+2,KP+1)+1.
      DO 22 I = 1,M+2
        A(I,KP+1)= -A(I,KP+1)
22     CONTINUE
20     IS=IZROV(KP)
      IZROV(KP)= IPOSV(IP)
      IPOSV(IP)= IS
      IF(IR.NE.0)GO TO 10
30     CALL SIMP1(A,MP,NP,0,L1,NL1,0,KP,BMAX)
      IF(BMAX.LE.0.)THEN
        ICASE = 0
        RETURN
      ENDIF
      CALL SIMP2(A,M,N,MP,NP,L2,NL2,IP,KP,Q1)
      IF(IP.EQ.0)THEN
        ICASE = 1
        RETURN
      ENDIF
      CALL SIMP3(A,MP,NP,M,N,IP,KP)
      GO TO 20
      END
```